AFRL-IF-RS-TR-2001-286
Final Technical Report
January 2002

MODEL-INTEGRATED COMPUTING ENVIRONMENT

Vanderbilt University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D937

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-286 has been reviewed and is approved for publication.

APPROVED: ROGER DZIEGIEL, Jr.
Project Engineer

FOR THE DIRECTOR: MICHAEL L. TALBERT, Maj, USAF
Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Jan 02	Final Jul 96 - May 01

4. TITLE AND SUBTITLE

MODEL-INTEGRATED COMPUTING ENVIRONMENT

5. FUNDING NUMBERS

C - F30602-96-2-0227
PE - 62301E
PR - D937
TA - 01
WU - 01

6. AUTHOR(S)

Janos Sztipanovits, Gabor Karsai and Akos Ledeczi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Vanderbilt University
PO Box 1829B
Nashville, TN 37235

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTD
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2001-286

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Roger J. Dziegiel, Jr., IFTD, 315-330-2185

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Model-Integrated Computing (MIC) was developed for building embedded software systems. The key element of this approach is the extension of the scope and usage of models such that they form the "backbone" of a model-integrated system development process. In Model-Integrated Computing models play the following central roles: 1) Models can explicitly represent the designer's understanding of the entire system, including the information processing architecture, the physical architecture, and the environment it operates in. 2) Integrated modeling allows the explicit representations of dependencies and constraints among the different modeling views. 3) Tools analyze different, but interdependent characteristics of systems (such as performance, safety, reliability, etc.). Tool-specific model interpreters translate the information in the models to the input languages of analysis tools. Using MIC technology one can capture the requirements, actual architecture, and the environment of a system in the form of high-level models. The requirement models allow the explicit representation of desired functionalities and/or non-functional properties. The architecture models represent the actual structure of the system to be built, while the environment models capture what the "outside world" of the system looks like. These models act as a repository of information that is needed for analyzing and generating the system.

14. SUBJECT TERMS

modeling, model integrated computing, meta-programmable, model editors

15. NUMBER OF PAGES

40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

Model-Integrated Computing Environment Project Executive Summary

The Model-Integrated Computing Environment (MIC) project of the EDCS program has focused on developing the tools and techniques for supporting model-integrated system development. The project has concentrated on the following areas of activities:

1. Meta-programmable model builder tool. The objective of this research direction was to develop the technology for meta-programmable model editors. The larger goal was to provide a reusable technology for developing domain-specific modeling environments that could be easily customized for specific domains. Specific results in this activity include:
    - Two generations of meta-programmable model editors: one for the Unix platform with object-database support, and another one for the Windows platform with repository support)
    - Two generations for meta-programming: one for Unix platform using a meta model specification language, and another one for the Windows platform using a graphical technology based on the industry standard UML notation
2. Specification of model interpreters. The objective of this research direction was to develop technology for the specification and rapid development of model interpreters (a.k.a. generators). Specific results in this activity include:
    - Multiple designs for the specification language documented in a various papers.
    - Practical implementation of one of the approaches using the visitor/traversal approach.
3. Run-time systems. The objective of this research direction was to evaluate, revise and enhance the technology for run-time systems of model-integrated systems. Specific results in this activity include:
    - A revised and enhanced Multigraph Kernel (MGK) which implements forward propagation in the dataflow graph and optimized for performance.
    - An experimental evaluation of the MGK on a multiprocessor platform.
4. Applications. The objective of this research direction was to demonstrate the applicability of the research results in practical systems. Some of these practical systems were jointly funded by external entities (e.g. USAF Arnold Engineering Development Center, The Boeing Company, Sandia National Laboratories,). Specific results in this activity include:
    - *Saturn Site Production Flow.* It is a model-based system for plant monitoring, data archival, and distribution used at GM's Saturn facilities.
    - *Integrated Test Information System.* It is a model-based system for test information system configuration and management used at the USAF Arnold Engineering Development Center and other places.
    - *State-Space Analysis Tool.* It is a model-based system for the design-time analysis of high-consequence systems used at Sandia National Laboratory.
    - *Diagnostics Tool Integration Framework.* It is a model-based tool integration framework for Prognostics and Health Management (PHM) tools used by Boeing on their military aircraft programs.

The application systems developed are based on the DARPA funded toolset, and are either in daily use by practicing engineers, or are being migrated into practical use.

This final report summarizes the progress and results achieved in the four specific areas mentioned above. The Appendix contains various published papers and other project documents.

1. Meta-programmable model builder tool

In a model-integrated development process, modeling and model building play a key role [1]. These steps in the process are supported by a model editor tool, which not only provides a visual interface to the models, but also performs semantic checks on those in order to ensure that the models are correct (with respect to some statically defined semantics). This model editor tool must be customized for the specific application domain. In order to make this process affordable (and most of the model editor's code reusable), we have chosen a meta-level approach: the generic model editor can be configured and customized for specific domains. This process is called "meta-programming of the editor" or "meta-modeling" [1].
We have developed two generations of the meta-programmable model editor: one for the Unix platform, and another one for the Windows platform. The capabilities of the two systems are summarized below. The two systems are called VME and GME, respectively.

The Visual Model Editor for Unix (VME)

The meta-programmable Graphical Model Builder (VME) is a visual diagram editor that can be adapted to a wide variety of modeling paradigms. The editor supports

- Visual diagram editing of models that contain icons, connections and comments
- Hierarchical diagrams
- Reusability through types and instances
- Model-database browsing
- Support for managing large-scale databases
- Multiple aspect editing
- Conditional zed diagrams
- Remote references to models
- Constraint management

VME's architecture is shown on Figure 1. It consists of two components: a database manager (to coordinate the simultaneous access to the model database) and the editor itself. The database manager runs as a background process that is shared by multiple editor processes. Editor processes maintain communication with the database manager and request authorization for database updates. At any time, there is only one editor doing updates.

The editor process supports database browsing and model editing. The databases browser component facilitates:

- Project browsing (in the form of projects, phases, and partitions), and
- Database partition browsing (this is where models are listed).

The model database is segmented into projects, which are segmented into phases, which in turn are segmented into partitions. A database partition is a portion of the database that can be accessed and updated independently (i.e. the unit of transactions on the database is the partition). It contains the models (types and instances), and explicit indication of models that are exported (for use in other partitions) and models imported (from other partitions).

The model editor component provides a diagram editor which supports visual model editing. The visual editing interface is completely replaceable: currently it is implemented in X/Motif, but an experimental Java interface has also been built and demonstrated.

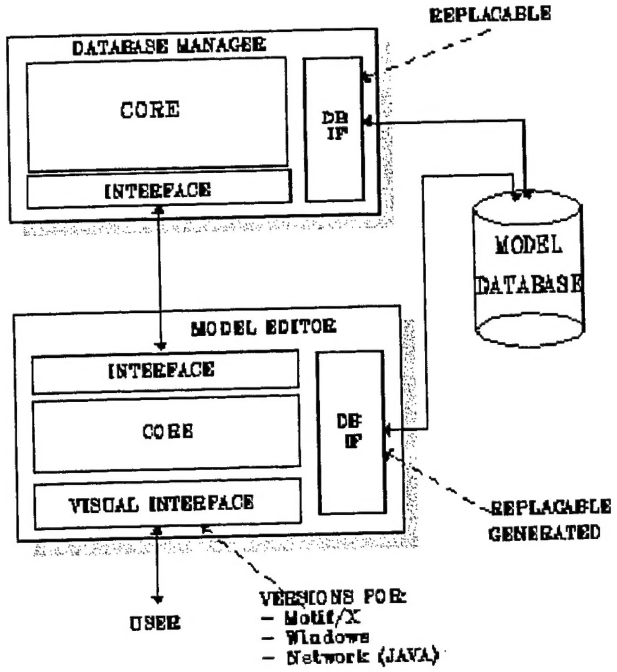

Figure 1: VME Architecture

Figure 2 shows an example screenshot of VME as being used to edit models of a signal processing modeling domain.

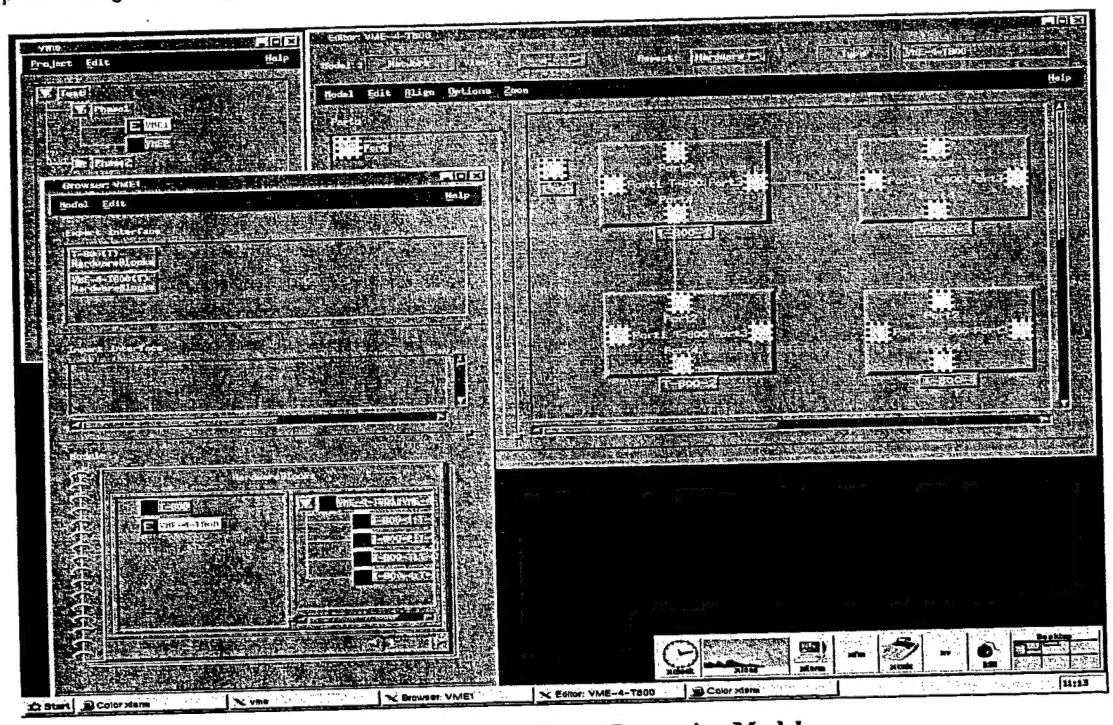

Figure 2: Example Signal Processing Model

Figure 3 shows VME used in the FDIR (Fault Detection Isolation and Recovery) modeling of the Space Station.

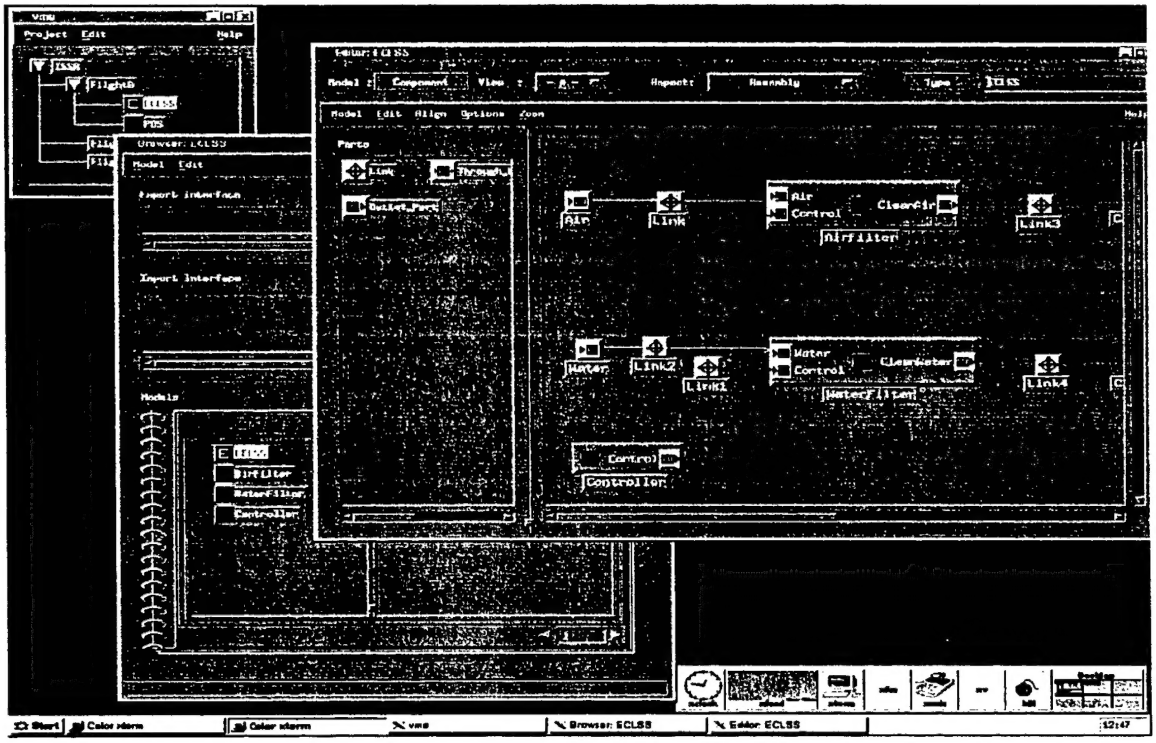

Figure 3: Example model from the Space Station ECLSS

For model storage an object-oriented database is used: two versions are available one in OBST[1] and one in Objectivity[2]. The model editor uses a separate database interface component to communicate with the database package

Meta-programmability means that components of a VME installation are generated/configured from meta-level descriptions. In particular, the database interface of the manager and of the editor component is generated, while the core of the editor is configured through the use of a configuration file. The database interface in the editor has the task of mapping domain- (and database package-) specific model objects into generic model objects used in the editor, and translating the editing visual operations into operations performed on the database.

Meta-programming is implemented via meta-level translators (one for OBST, one for Objectivity). The input to these translators is an MDF file, which is then used to generate the database schema, the database API, and the interface code for the editor and the database manager. The section labeled Appendix A.1 contains the MDF specification for a simple signal processing application.

Boeing has used VME on their Space Station program to produce models for the FDIR[3] work. The complexity of the models was stretching the underlying database implementation technology: the models contained components on the order of thousands. The system has also been used on other projects like the SSAT project with Sandia, the Telescience project with Boeing, and the SSPF project with Saturn [2].

---

[1] OBST is a public-domain object-oriented database developed at FZI, University of Karlsruhe, Germany.

[2] Objectivity is a commercial object-oriented database, available from http://www.objectivity.com

[3] Fault Detection, Isolation, and Recovery: activity of the Space Station design where fault models of the system are built and analyzed to determine the diagnosability properties.

The Generic Modeling Environment for Windows (GME)

Because of the increased availability of high-end Windows platforms, it was necessary to re-engineer and reimplement the meta-programmable modeling environment for that platform. The result of this work is the GME [3]. GME is a new generation of environments optimized for the Windows platform. Figure 4 shows the architecture of the package.

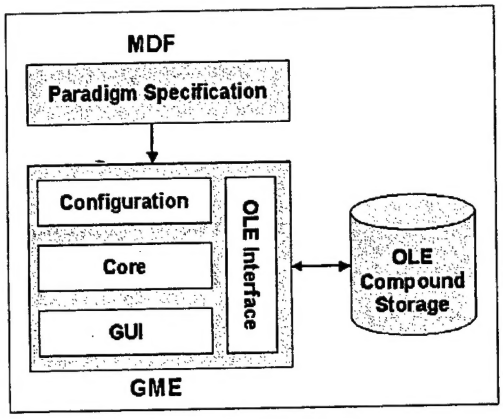

Figure 4: GME Architecture

GME is composed of a few major components that were integrated using Microsoft's COM integration technology. It is using persistence facilities provided by Microsoft: OLE storage (for the low-end applications) or MS Repository (for the high-end applications). Figure 5 shows an example GME screen illustrating its usage on a signal-processing example.

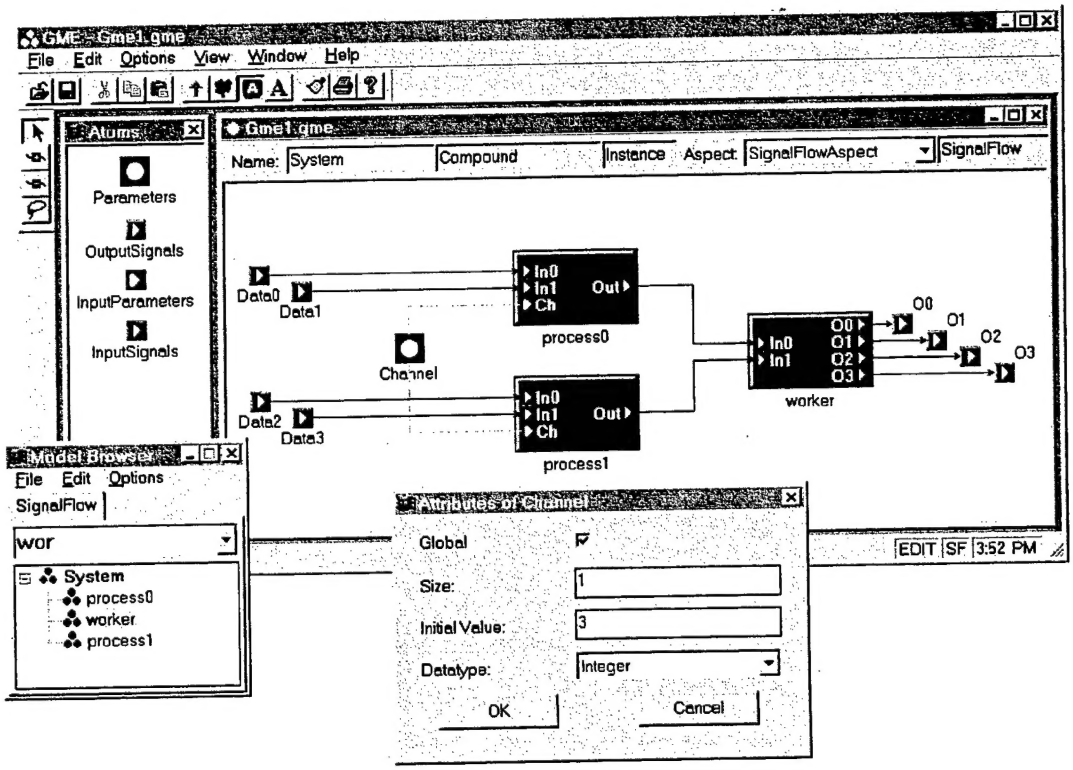

Figure 5: Example GME screen

In addition to the reengineering of the environment, the major result of the work is a new approach to meta modeling. We have studied formal meta-modeling approaches [4], and the conclusion was that the meta-modeling has to be formalized and should not rely on an ad-hoc meta-modeling language (like MDF). The first version of the meta-modeling approach was based on the following concept: the meta-models were the graphical specifications for the contents of the MDF files. This has been found difficult to use, so we have developed a new technology based on using UML and OCL for meta-modeling [5]. UML class diagrams are used to capture the concepts and their associations in a modeling domain, and OCL constraints are used to specify the static semantics [6]. The meta-modeling approach is an application of the well-known four-layer architecture [21]. Figure 6 shows an example GME metamodel.

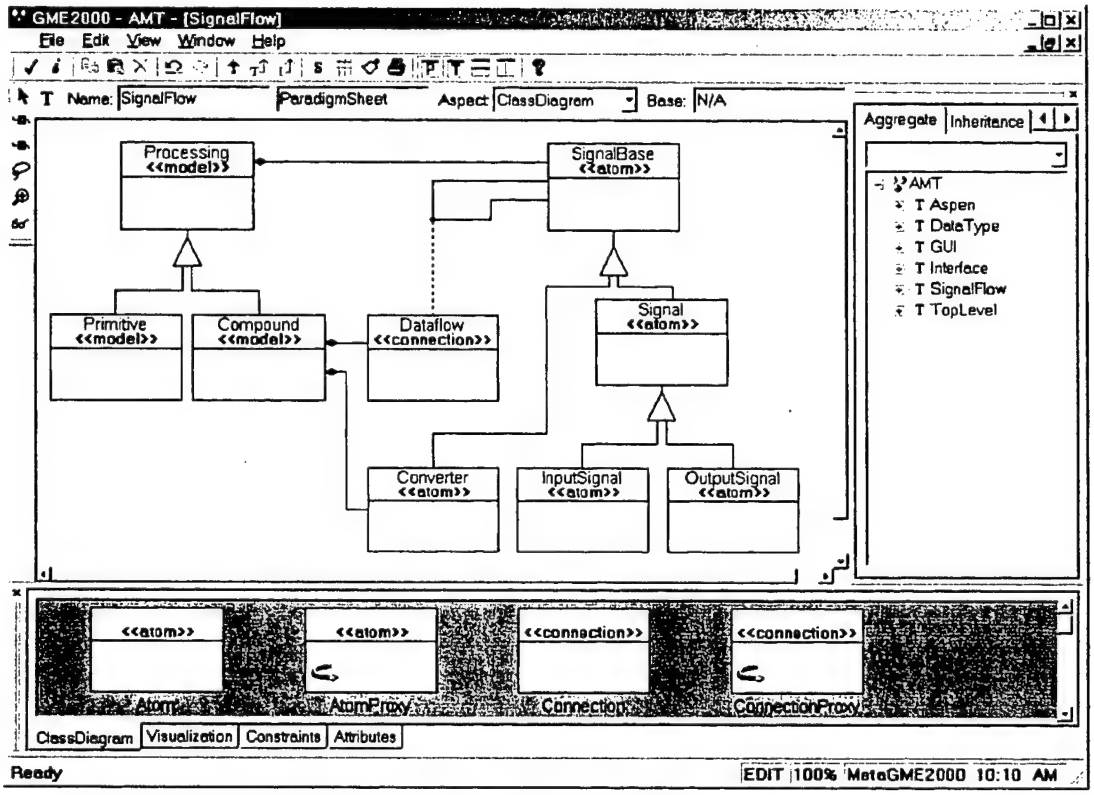

Figure 6: Example GME metamodel

The GME environment (and its newer release GME 2000) has been in active use since its development in all of our projects that need graphical modeling capability. The active websites [7] and [8] contain the most update information about this tool. GME supports all the modeling techniques developed for VME, although some of the techniques have different names. For further details, please see the websites mentioned above, and the documentation included in the GME distribution.

As conclusion for further research, we have to emphasize the need for high-performance, yet flexible modeling environments. The chosen meta-modeling approach (UML and OCL) has been proven successful, but the meta-model diagram complexity limited the sophistication of the modeling paradigms that could be defined. We recommend further research work in meta-modeling, to help managing complexity. Visualization techniques have to be improved as well, and integrated with non-iconic techniques. In GME, the current visualization primarily relies on ported objects, but other diagrammatic, tabular, and textual schemes must be supported as well.

2. Specification of model interpreters

In a Model-Integrated Computing Environment the model interpreters perform the mapping between the domain-specific models and the executable models of the run-time environment or the analysis models [1]. The research in this task has focused on the specification of model interpreters using higher-level techniques and prototype implementations.
The first results here were used in the Unix version of the meta-programmable model builder, VME. We have built a prototype meta-level translator that transformed specifications written in a meta-language into actual code that became a component in the MIPS environment. Figure 7 below illustrates how the translator operates.

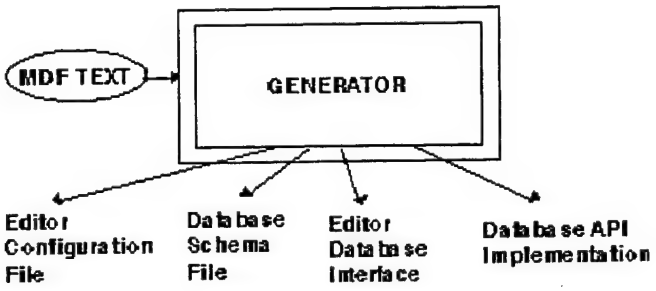

Figure 7: MDF meta-translator used in VME

The input to the generator was in the form of a MDF (Model Definition Facility) file that captured relevant information about the modeling domain, including modeling concepts, the model organization techniques used, and domain-specific constraints on the models. The generator generated various source and other files, including an editor configuration file a database schema specification (in the database schema language of OBST and Objectivity) an editor database interface program (used in the visual model editor to interface the generic core of the program with the domain-specific model database) an implementation of class methods of persistent objects (i.e. a database API). This technology has been successfully used in the visual model editors, but it was restricted to that particular application.

The translator for this case was primarily hand-crafted, but it served as a useful learning tool for defining complex model interpreters. One lesson we have learned was that the generators always have to traverse a data-structure, take actions while visiting specific nodes, and then produce some output. The most difficult task in writing an interpreter is to code these traversals, and to maintain the mapping between the input and the generated outputs. The task of model interpreters is typically easier than that of compilers (e.g. model interpreters rarely deal with code optimization issues), and the task of the interpreter writer is often specialized to the needs of the particular application. Where the specification and automatic generation model interpreters should help in is the mundane tasks associated with the traversals and bookkeeping.

To extend the usability of the generators for other domains (not only editors) and to generalize the approach, we have developed a new technique based [23] on the Visitor pattern [23] and the Adaptive Programming [23] techniques. The technique is based on a *traversal language* that allows the expression of traversal sequences performed by the model interpreters. Figure 8 shows the architecture or model interpreters as implied by the technique.

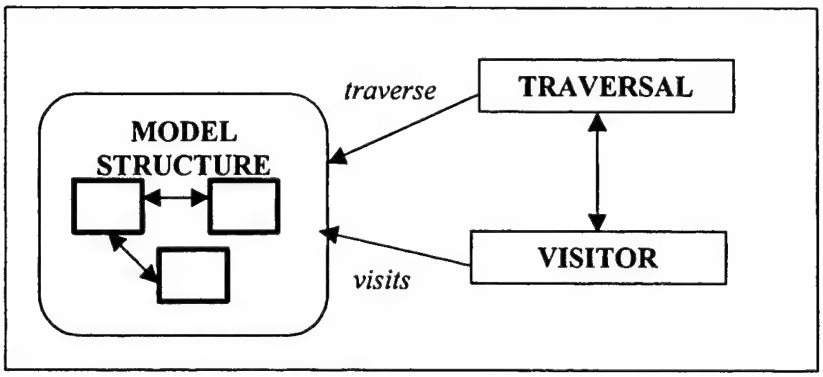

Figure 8: Model interpreters using traversals and visitors

Here the model interpretation process is considered as (1) executing a *traversal* on the objects of the model structure, and, at each object (2) executing a *visit action*. The traversal is a formal specification of what nodes of the graph to visit and in what order. The visit actions are computations executed when visiting a node. The traversal language we have defined allowed the concise specification of traversal and visitor objects, together with procedural code in C++. This mixed-mode programming technique has proved very practical: the high-level constructs were very expressive to capture the traversal sequences, while the low-level action code was efficient in expressing procedural computations. Appendix A.2 shows the full code for a model interpreter.

The model interpreter traversal language was processed by a meta-translator [23], which generated the actual C++ code for the model interpreter. This meta-translator had two inputs: a specification of the data-model of the model structure (practically, the meta-model of the modeling language), and the traversal specification itself. The translator parsed its input and created abstract syntax trees from them, and then generated the full executable code for the model interpreter. Embedded C++ code (shown between << and >> in the example) was literally copied to the output, while the traversal specifications were translated into procedural code that traversed the input data structure using C++ iterators and recursive descent techniques. The traversal and visitor specifications were transcribed into two C++ classes, with appropriate methods for capturing the various actions. The two classes were arranged to form a coroutine-like control structure: control passed between the two classes in an alternating manner. We have introduced several extensions into the traversal specification language that allowed us to write multiple pass interpreters that keep a context during the traversal.

The model interpreter generation technique has been extremely successful in one of the application projects: the PHM tool integration (discussed in a section below). We were able to develop new model interpreters (called "semantic translators" in that context) very rapidly. A typical model interpreter was dealing with modeling concepts on the order of tens (~20 in the average case), and translating them into other concepts, again on the order of tens. The typical model interpreter took a few pages, written in a mixture of the traversal language and C++ code, and the generated code was about three times as large. On the average, it usually took a man-week to develop one model interpreter using the technique described.

Regarding further research in the area, we recommend the investigation of higher-level techniques for modeling the model interpreters. This would allow the more formal specification of model interpreters, in a way such that correctness of the interpretation scheme could be proven using automated tool support. Model-integrated environments have a crucial component in the interpreters: their performance (in terms of the quality of the generated systems) determines the success of the environment.

3. Run-time Systems

In a Model-Integrated Computing Environment, specialized run-time systems (RTS) serve as the execution platform. The RTS provides services for facilitating component interactions and act as a run-time integration framework in which the components live.

Based on previous research results, we have refined and re-implemented a run-time infrastructure that we have found very useful for many model-integrated application domains: the Multigraph Kernel (MGK). MGK implements an asynchronous dataflow model of computation, where components are ported objects. In MGK terminology they are called actor nodes, they have an associated computational procedure called the script, and they can also have local state, called the context. The actor nodes are scheduled by the MGK scheduler according to the dataflow principle: a node is executed when data is available on its input ports. When the node runs, it generates data that is sent out via its output ports. Data is propagated through the graph, and nodes fire upon the arrival of data. This model was used in building many model-based applications, including signals processing, control, and others.

MGK extends the above basic dataflow model in several ways:

- **Extended connectivity**: MGK allows every node input and output to be connected to several other nodes. This simplifies the writing of the node operation procedures (the *scripts*), as these will have to process and generate only the functionally different input and output data, while any data stream merge or replication is handled at the dataflow kernel level.
- **Flexible node scheduling**: MGK allows nodes to be triggered in different modes. It is possible to run a node when all of its inputs are present (*ifall* triggering principle), when any single input is present (*ifany* triggering principle) or in the presence of user-defined subsets of the full input data set (*custom* triggering). Additionally, node scheduling is priority based.
- **Dynamic control graph building and reconfiguration**: MGK applications are built at run-time using the kernel's API functions. This makes MGK especially suited as the target environment of the model-based MA model interpreter tools. Additionally, MGK also allows changes on an already executing dataflow control graph, making dynamic reconfiguration of running applications possible.
- **Local memory for processing nodes**: The MGK script interface API allows the creation of reusable, customizable node operation primitives. Besides providing a unified mechanism to retrieve and propagate data in the control graph, the MGK script API also allows the creation, filing and retrieval of local node parameters (called the node's *context*). The availability of the node context makes it possible to customize more generic scripts to a certain function in the graph (i.e. fewer scripts will have to be written). The context can also be used to store data between successive executions of the node.
- **Dynamic data typing and memory management**: Various data types can be propagated in MGK control graphs. These include the usual scalar types (integers, floating point numbers, etc..) and blocks of memory containing user-defined structures. The kernel provides its own memory management services that also include garbage collection.
- **Flow control and exception handling**: Every MGK node connection can have an attribute that specifies the maximum number of buffered data items. Exceeding this limit generates an exception. These exceptions, along with other kernel-generated events, can be trapped by user-defined handlers. In case the user does not wish to trap certain exceptions, the kernel provides reasonable default handlers.
- **Support for distributed parallel architectures**: MGK transparently supports the creation of dataflow control graphs in distributed environments. These include networked UNIX and 32 bit Windows platforms, running either TCP-IP or some other message-passing environment (e.g. MPI). Certain digital signal processing (DSP) processors with

on-chip communication capabilities (Texas Instruments C40, C44, etc..) are also supported.

MGK has been reengineered and enhanced to support a simpler API. [12] contains the full user's manual for the improved package. Appendix A.3 contains the full code for a simple MGK application.

MGK has been used in numerous applications and platforms, ranging from high-performance DSP-s, through Windows PC-s, to high-end Unix workstations. Figure 1 shows an example for an application where MGK was used as the run-time platform. The application was the instrument control for a Telescience experiment of Boeing used in their Space Station work. In the research work, we have also ported and evaluated MGK on a high-end parallel workstation platform: the SP2 system of IBM. The results have been documented in the report [13].

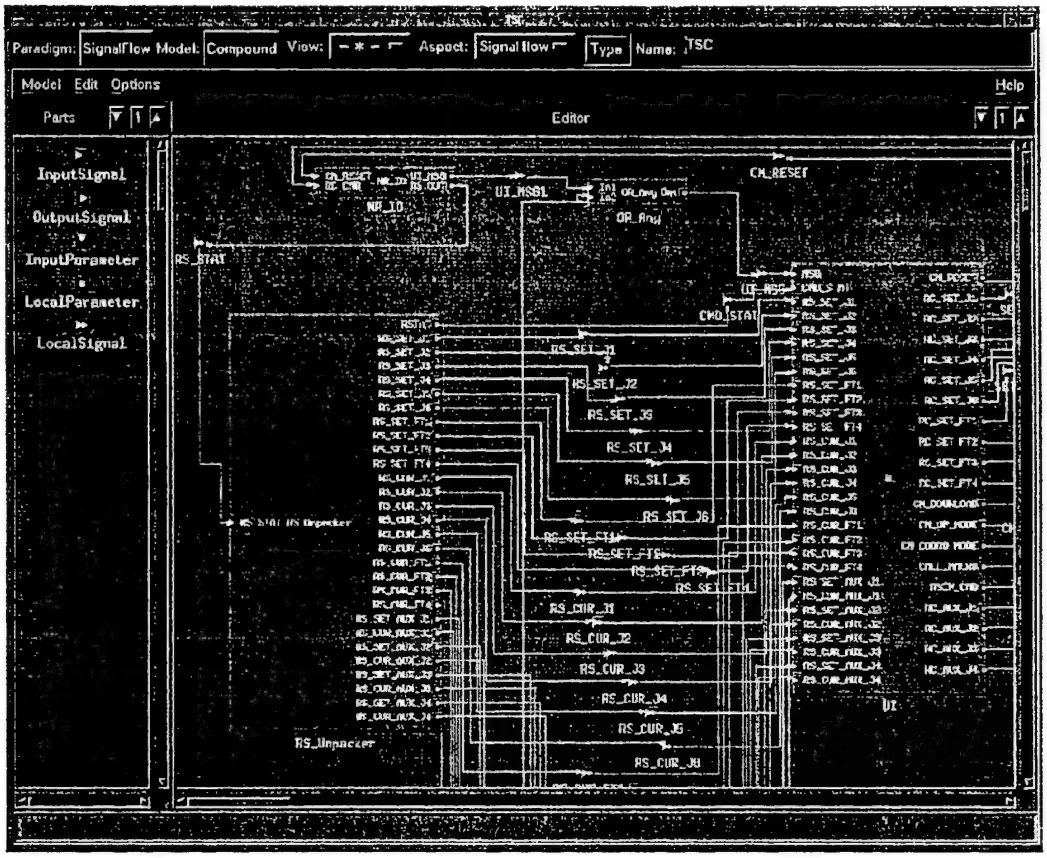

Figure 9: Example application model that has used MGK as the run-time system

Regarding further research, we recommend the investigation of other models of computations for run-time systems. While the asynchronous dataflow model used in MGK was very useful, it is not the only one, and other models and their requirements for run-time support has to be understood. We envision that, especially in embedded information systems, application require a wide variety of run-time support, and these subsystems must interact with each other. Integrating multiple models of computations in a coherent framework (where all behaviors can be precisely described and predicted) is a difficult problem, but it has to be addressed if the model-integrated approach is to be applied in other applications.

4. Applications

We have used the technology and tools described above in a number of real-life applications. In this section a short summary of major applications is provided. These projects were funded by third parties: General Motors, Corp, USAF Arnold Engineering Development Center, Sandia National Labs, and The Boeing Company. However, all of the projects have used the tools technologies created in the course of the EDCS DARPA project.

Saturn Site Production Flow (Saturn Plant, GM)

SSPF [14] is a large-scale business application used in the Saturn facility of General Motors in Spring Hill, TN and in Wilmington, DE. It collects, manages, archives, and serves plant production data for the entire manufacturing process. It is in daily use since 1997, with continuing upgrades. The problem solved by SSPF is as follows: The Saturn plant had a data acquisition system to monitor the discrete manufacturing processes, but higher-level processing, distribution, and management of data was lacking. Data had to be archived into large-scale product data archives, and it also had to be distributed, in real-time, to hundreds of workstations across the plant. Furthermore, minor (on the average, bimonthly) and major (yearly) configuration changes in the plant necessitated the timely reconfiguration of the information system. Additionally, the plant engineers needed various data simulation and analysis tools for identifying problems with the production processes (e.g. bottlenecks).

We have built SSPF using the MIC technology developed in the course of this research. The approach was applied as follows.

- We have designed a modeling language to capture the structure and organization of manufacturing production processes, and other relevant information (e.g. data interface point names, calculation algorithms, GUI screen layouts, etc.)
- We have developed model interpreters that generated configuration files and source code for the generic the run-time system components from the models.
- We have selected off-the-shelf components for the run-time system (e.g. SQL Server for database, the Simplicity data acquisition package, etc.). We have also developed generic components for those cases where COTS solutions were not available (e.g. data viewer clients).
- We have integrated the system using a network-oriented integration technology, and trained plant engineers and technicians how to use it.

The resulting system allows the rapid model-based configuration (and re-configuration) of the information system, which is a front-line system at a major manufacturing facility. The resulting monitoring system: a site production flow application has been installed and is in daily use at two plants. The system monitors a few thousand data points and broadcasts real-time data to about 300 workstations for inspection. Saturn estimated that the system can be credited 2-5% throughput increase - which is about 25-60 cars per day. The second installation demonstrated the power and flexibility of the model-integrated approach: the plant was modeled (2 man/weeks effort), and a running system was created in one day. It is also extremely easy to maintain: when the plant changes, the visual models are modified, and the entire system regenerated in a matter minutes. Plant modeling is done by trained technicians, which proves that the model-integrated systems are end-user programmable to a high degrees. Figure 10 shows an example client screen displaying production data. There were several lessons learned from the project [15] that influenced our technology development as well.

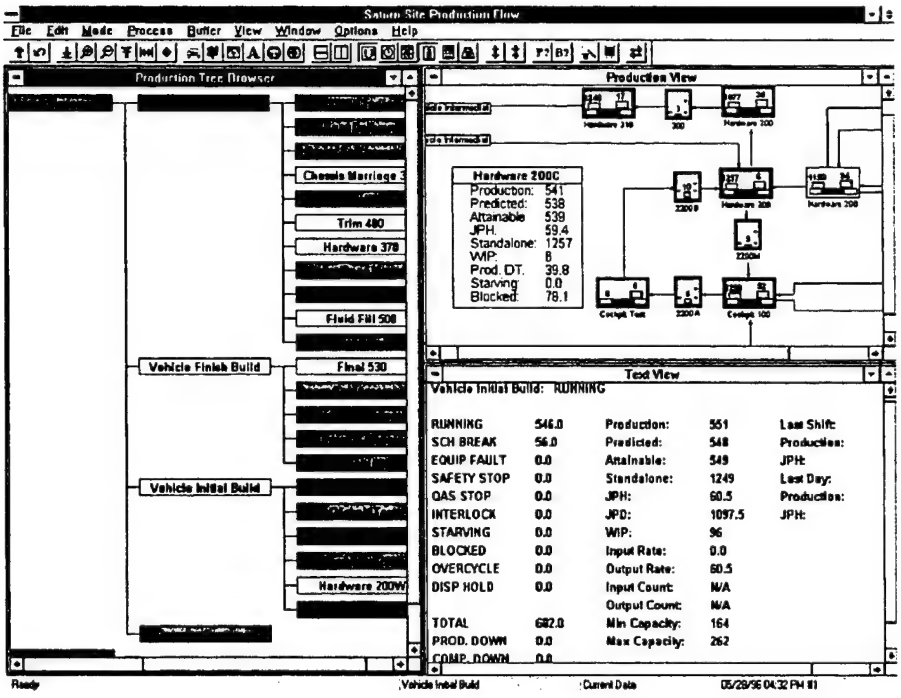

Figure 10: SSPF GUI Screen

Integrated Test Information System (USAF AEDC)

ITIS has been developed for the Arnold Engineering and Development Center of USAF. AEDC is the primary aerospace engine testing facility of the nation. The main product of AEDC is test data that has to be produced, processed, and delivered to customers on a timely basis. This process is supported by a complex test information system, whose management and configuration is a highly non-trivial task. In the past, it was not unusual to spend man-months of effort just on reconfiguring the information system for the needs of a particular test.

The objective of ITIS (see Figure 11) was to automatically synthesize complex test information systems using high-level system models. The ITIS makes it possible to automatically configure the individual test systems in a large-scale aerospace test facility, and the generic information server (ITIS Server). The test systems include: simulations systems, data analysis systems, real-time test data systems, archived data systems, and pricing data systems. The ITIS server gathers data from the test systems and makes that available on the World Wide Web. Clients can connect to the server and visualize the data using JAVA technology. Using the model-integrated technology, the system can manage rapidly changing requirements for the interconnection of a wide variety of data sources and analyses.

ITIS helps managing and distributing test data, whose timeliness is of utmost importance. The reduction in design/test cycle time enabled by this project cut the systems development cost by tens of millions of dollars per year[4]. Pratt and Whitney stated that ITIS is "an essential part of our plans for support of aeromechanical testing" for the JSF engine. "ITIS has been successfully demonstrated to be an effective monitoring technique for strain gage test support. This allows engineers to watch testing at their desks at Pratt & Whitney Florida and East Hartford campuses. By enabling an immediate picture of an ongoing test, risk can be reduced by involving experts in the current issue by a mere click of the mouse at their PC." The ITIS web-based application and

[4] Source: USAF AEDC.

modeling tools were installed at the Air Force SEEK EAGLE Office (AFSEO) on Eglin Air Force Base in Ft. Walton Beach, Florida, and at the St. Louis facilities of the Boeing Company. AFSEO is a customer of AEDC and wishes to adopt the same infrastructure for sharing test metadata and data. The Air Force Information Warfare Center (AFIWC) staged an "attack" on the ITIS application to verify it meets Air Force security requirements, and the system passed all tests. The attached documents [16] and [17] provide technical details about the system.

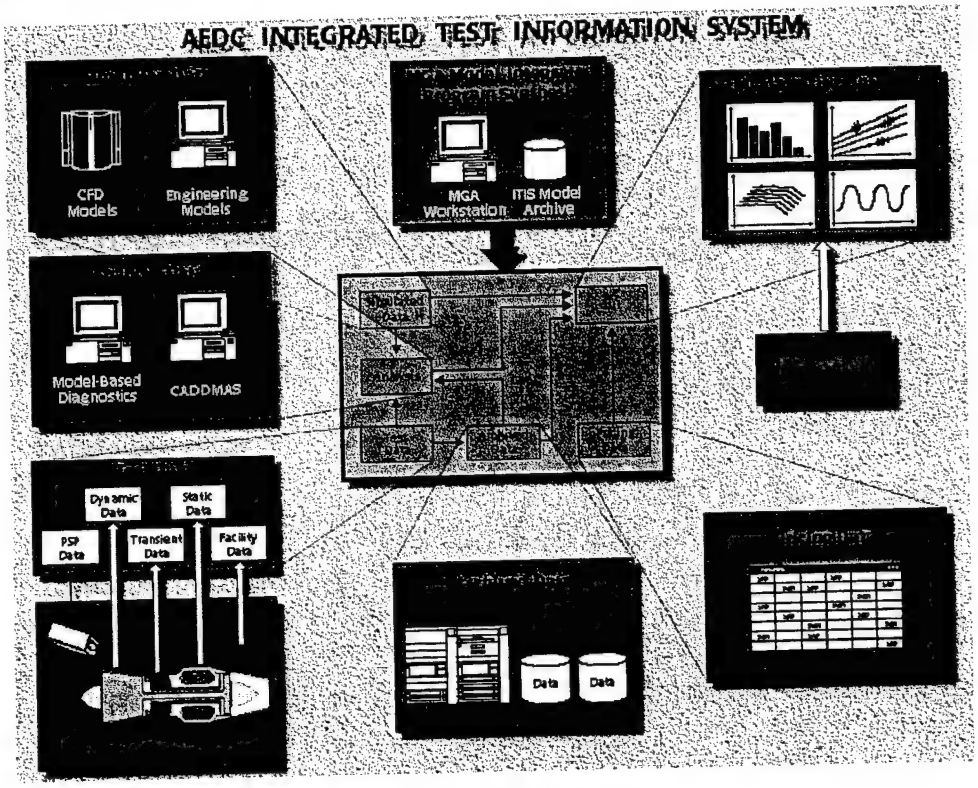

Figure 11: ITIS Architecture

State-Space Analysis Tool (Sandia National Labs)

SSAT is a design analysis tool for the joint reliability, safety, and performance analysis of high-consequence systems. Examples for high-consequence systems include automotive breaking systems and special weapons. Using traditional approaches, the performance, safety, and reliability aspects of system designs are considered independently. Changes made in one aspect of the design interact with the other aspects, making the design process very complicated and hard to manage. In practice, the conflicting aspects of the design have to be evaluated simultaneously.

SSAT that has been developed for Sandia National Labs addresses this problem by applying a model-integrated approach. In order to model high assurance, high consequence systems, a modeling paradigm has been developed to capture critical safety and reliability information about the system. Parallel, hierarchical, finite state machines are used to model high-level system behavior. These same types of models are used for capturing the system's hardware behavior. We have defined a modeling language to capture *both* the nominal and fault behavior of the system using a set of integrated models. From this detailed description of system level and hardware behaviors, safety and reliability analyses can be performed.

From the models we generate multiple representations that are suitable for various analyses. One type of representation represents the system as a complex, hierarchical finite-state machine using Ordered Binary Decision Diagrams (OBDD-s). OBBD-s are also used in the semiconductor industry for verifying logic designs of high complexity. We have developed an analysis tool that performs full state-space exploration on the design, verifying, for instance, safety requirements. "Safety" has been defined as: "the system will never reach an unsafe state, even under fault conditions". Other analysis algorithms include: forward system simulation, backward system simulation, forward and backward reachability analysis, deterministic analysis, loop analysis. Another representation generated from the models is fault-trees that are processed by off-the-shelf reliability analysis tools. The objective here is to calculate overall system reliability from component reliability data and system structure. We have interfaced our environment with an off-the-shelf (WinR from Sandia National Labs) to support this task.

This integrated toolset [18], consisting the modeling environment, the tool for the analyses, and the automatic fault tree generation component comprises the SSAT toolset. The tools have been used in the modeling and analysis of automotive breaking systems and on the analysis of the design of the W-88 warhead. The tools are currently in daily use by design engineers at Sandia. Figure 12 shows the user interface of the analysis tool.

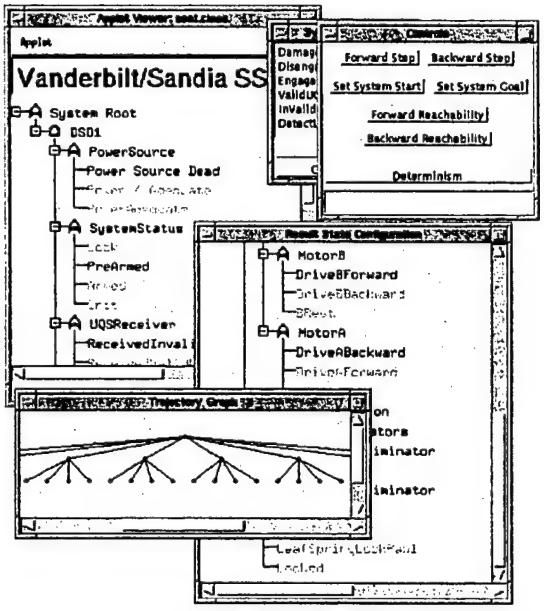

Figure 12: SSAT Analysis Tool

Diagnostics Tool Integration Framework (The Boeing Company)

Engineering processes usually employs a number of design tools for analysis, data capture, simulation, etc. These design tools all contain data related to the same physical artifact (e.g. an aircraft design), but they are usually very poorly integrated. Data is often replicated, hand-translated between the tools, and its management is very difficult. This problem has come up at the Boeing Company in one of their major military aircraft programs, in the Prognostics and Health Management area.

We have developed semantic integration framework [20] [19] for engineering tool integration using the model-based technology. This framework lets the system integrators model the

engineering tools and create custom integration solutions for interoperating CAD tools and databases. The integration framework (see Figure 1) relies on a client/server architecture, where the design tools are equipped with tool adaptors that are the clients of the integration server. The tool adaptors are responsible for the syntactic transformation of the tool data between the tool and a canonical form, and exchanging data in that canonical form via a network protocol. The integration server contains semantic translators that transform data between the schema used in the tools and an integrated schema. Data expressed using the integrated schema is stored in a database inside the server. The key component in this architecture is the semantic translator, and this is the place where the model-integrated technology has been used. The translators have been created using the techniques we have developed for specifying and generating model interpreters. The input and the output of the translators are modeled (in the form of UML-like meta models). The mapping between the two has been expressed using the language we have developed for representing model interpreters. From these three models we have automatically generated the code of the translators.

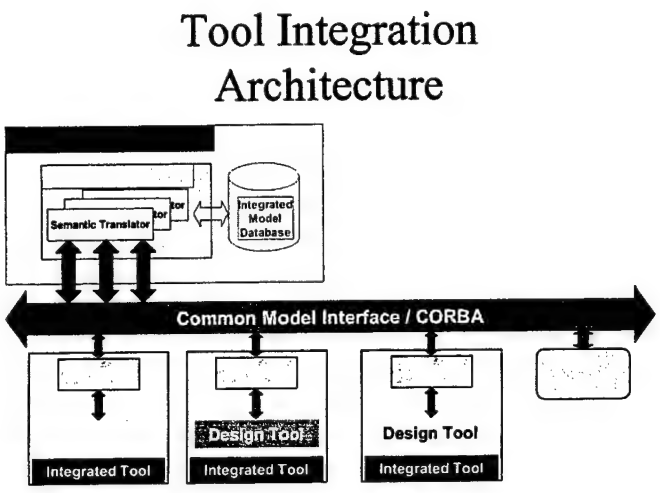

Figure 13: Tool Integration Framework

We have built two integration framework instances: one for five tools: AEFR (a diagnostics expert system), an FMECA database, GME/AVDR (a diagnostics modeling environment), Relex (a reliability analysis package), and Advise (a testability analysis package), and another one for three tools. The advantage of the model-integrated approach was obvious: it has greatly simplified the system evolution and the development of translators. The approach and the tools have also been demonstrated to DoD SPO personnel at several TIM-s where they received high marks.

Appendix

A.1 Example MDF file for representing a signal flow language

```
// xdl.mdf
paradigm XDL;
attribute Description : page "Enter model description:" (8 40) "";
attribute Datatype : menu "Select datatype"
                   { "Int"    Int_type ;
                     "Float"  Float_type ;
                     "Double" Double_type ;
                   };

atom Port { icon "port.icon"; }

modelclass MultiProc {
  Hardware {
    attrs {attr Description;}
    parts {Ports : Port link;}
  }
}

model Node is_a (MultiProc) primitive {
  Hardware "Hardware" {
    icon rect; font 3; color foreground;
    attrs {Memory : field int "Memory size:" "1";}
  }
}

model Network is_a ( MultiProc ) compound {
  Hardware "Hardware" {
    icon rect; font 3; color foreground;
    conns {
      HWConn { 1 solid line butt } :
      { Ports -> NetworkParts Ports }
        { NetworkParts Ports -> NetworkParts Ports }
      { NetworkParts Ports -> Ports }
        { NetworkParts Ports -> NodeParts Ports }
      { NodeParts Ports -> NetworkParts Ports }
        { Ports -> NodeParts Ports }
      { NodeParts Ports -> NodeParts Ports }
      { NodeParts Ports -> Ports };
    }
    parts {
    NodeParts: Node hierarchy;
    NetworkParts : Network hierarchy;
    }
  }
}

atomclass Signal {
  attrs { attr Description; attr Datatype; }
}

modelclass Processing {
  SignalFlowAspect {
    attrs { attr Description; }
    parts {
      InputSignals : InputSignal link  ;
      OutputSignals: OutputSignal link ;
    }
  }
  StateAspect {
```

```
    attrs { attr Description;}
    conns {
      StateTransConn { 2 solid line arrow } :
      { States -> StateTransitions };
      TransStateConn { 2 solid line arrow } :
      { StateTransitions -> States };
    }
    parts {
      States : State;
      StateTransitions : StateTransition link;
    }
  }
}

atom InputSignal is_a ( Signal ) { icon "isig.icon"; }

atom OutputSignal is_a ( Signal ) {icon "osig.icon"; }

atom LocalSignal is_a ( Signal ) { icon "lsig.icon"; }

atom State {
  icon "state.icon";
  attrs {attr Description;}
}

atom StateTransition {
  icon "strans.icon";
  attrs {attr Description;}
}

atom Script {
  icon "script.icon";
  attrs {Text : page "Script text:" (16 64) "";}
}

model Primitive is_a ( Processing ) primitive {
  views { Data_Flow User_Interface }
  SignalFlowAspect "Signal flow" {
    icon rect {left : InputSignals; right: OutputSignals; };
    font 3; color foreground;
    parts {
      TheScript : Script single;
    }
  }
  StateAspect "State model" {
    icon rect {  top : StateTransitions;};
    font 3;
    color foreground;
  }
}

model Compound is_a ( Processing ) compound {
      views { Data_Flow User_Interface }
      SignalFlowAspect "Signal flow" {
        icon rect {
          left : InputSignals;
          right: OutputSignals;
        };
        font 3;
        color foreground;
        conns {
          DataflowConn { 1 solid line arrow } :
            { InputSignals -> PrimitivePart InputSignals }
            { InputSignals -> CompoundPart  InputSignals }
```

```
            { LocalSignals -> PrimitivePart InputSignals }
            { LocalSignals -> CompoundPart  InputSignals }
            { PrimitivePart OutputSignals -> LocalSignals }
            { CompoundPart  OutputSignals -> LocalSignals }
            { PrimitivePart OutputSignals -> OutputSignals }
            { CompoundPart  OutputSignals -> OutputSignals }
            { SignalRefs -> PrimitivePart InputSignals }
            { SignalRefs -> CompoundPart InputSignals }
          attrs { attr Description; };
      }
      parts {
        LocalSignals : LocalSignal;
        SignalRefs -> { ProcessingBlocks : Compound :
                        SignalFlowAspect : LocalSignals };
        NodeRefs   -> { HardwareBlocks : Network :
                        Hardware : NodeParts };
        ProcessorRefs -> { HardwareBlocks : Network : Hardware };
          PrimitivePart: Primitive hierarchy;
        CompoundPart : Compound hierarchy;
      }
    }
    StateAspect "State model" {
      icon rect {
        top : StateTransitions;
      };
      font 3;
      color foreground;
      conns {
        StateSubStateConn { 2 solid line arrow } :
          { States -> PrimitivePart StateTransitions }
          { States -> CompoundPart  StateTransitions };
      }
      conds {
        StateDeps States :
          { }
          { PrimitivePart CompoundPart };
      }
      parts {
      PrimitivePart: Primitive inherited  hierarchy;
      CompoundPart : Compound inherited  hierarchy;
      }
    }
}

category HardwareBlocks { Node Network }
category ProcessingBlocks { Primitive Compound }
```

A.2 Example model interpreter specification

```
// Model structure specification
paradigm Xdl;

entity Port { }
entity InputPort  : Port { }
entity OutputPort : Port { }

model Block {
  part InputPort  inputs;
  part OutputPort outputs;
}

model Primitive : Block {
  attr string type;
}

model Compound : Block {
  part Block blocks;
  rel  Connection conns;
}

relation Connection {
  Port src * <-> Port dst *;
}

// Model Interpreter Specification
interpreter XdlInterpreter;
<< typedef long Wire; typedef long Node;
   typedef map<ID,Wire> PortMap;
   int newId()  { static int count = 0; return count++; }
   int mkWire() { return newId(); }
   int mkNode() { return newId(); } >>

visitor Visitor {
  at Port [PortMap& sMap]
  << if(sMap.find(self.Id())==sMap.end())
       sMap[self.Id()]=mkWire(); >>;
  at Connection [const Block_M* parent, PortMap &sMap]
   << Port_E *src = self.src(), *dst = self.dst();
      Block_M* srcBlock = (Block_M*)src->Parent();
      Block_M* dstBlock = (Block_M*)dst->Parent();
      if((srcBlock != parent) && (dstBlock != parent)) {
         int tmp = mkWire();
         sMap[src->Id()] = tmp; sMap[dst->Id()] = tmp;
      } else if(srcBlock == parent) {
         sMap[dst->Id()] = sMap[src->Id()];
      } else if(dstBlock == parent) {
         sMap[src->Id()] = sMap[dst->Id()];
      } >>;
  at Primitive [PortMap& sMap] traverse[sMap];
  at Compound [PortMap& sMap]  traverse[sMap];
  at Port [int& count, Wire wire, Node node]
     << printf("connect wire:%d to node:%d[%d]\n",
                wire,node,count);
        count++; >>;
  at Port [Node node, Wire wire, int& count]
      << printf("connect node:%d[%d] to wire:%d\n",
                 node,count,wire);
         count++; >>;
}

traversal Traverser using Visitor {
```

```
  from Block[PortMap& sMap]
       to { inputs[sMap], outputs[sMap] };
  from Primitive[PortMap& sMap] do Block[sMap]
   << Node node = mkNode(); int count;
        printf("node %d %s\n ",node,self.type()); >>
   to { << count = 0;>> I
          inputs[count,sMap[arg.Id()],node],
    << count = 0;>>
          outputs[node,sMap[arg.Id()],count] };
  from Compound[PortMap& sMap] do Block[sMap]
       to { conns[&self,sMap], blocks[sMap] };
}
```

A.3 MGK Example Application

```
/***
*** DEMO.C
*** Multigraph Kernel v 6.0 simple demo program
*** Copyright 1986-1997, Vanderbilt University. All rights reserved.
*** This copyright notice is included for precautionary purposes
*** and does not constitute an admission that the material to
*** which it has been affixed has been made public or otherwise
*** disclosed without restriction.
***/

#include
#include "mgk60.h"

static void sadd(void) {
  void *in1,*in2;
  mgk_data_type t1,t2;
  in1 = mgk_receive(0,&t1);
  in2 = mgk_receive(1,&t2);
  if(in1 && in2 && (t1 == T_DOUBLE) && (t2 == T_DOUBLE)) {
    double res = *((double *)(in1)) + *((double *)(in2));
    mgk_propagate(0,&res,T_DOUBLE);
  }
}

static void ssub(void) {
  void *in1,*in2;
  mgk_data_type t1,t2;
  in1 = mgk_receive(0,&t1);
  in2 = mgk_receive(1,&t2);
  if(in1 && in2 && (t1 == T_DOUBLE) && (t2 == T_DOUBLE)) {
    double res = *((double *)(in1)) - *((double *)(in2));
    mgk_propagate(0,&res,T_DOUBLE);
  }
}

int main(int argc,char **argv) {
  if(mgk_initialize(&argc,&argv) == E_SUCCESS) {
    mgk_nodep n1,n2,n3;
    double data1,data2,data3,*dp;
    mgk_data_type tp;
    int i;
    n1 = mgk_create_node(sadd,2,1,100,AT_IFALL);
    n2 = mgk_create_node(ssub,2,1,100,AT_IFALL);
    n3 = mgk_create_node(sadd,2,1,MGK_NODE_STOP_PRIORITY,AT_IFALL);
    mgk_connect_nodes(n1,0,n2,0);
    mgk_connect_nodes(n2,0,n3,0);
    data1 = 5.0; data2 = 2.0; data3 = 3.0;
    if(argc > 1) sscanf(argv[1],"%le",&data1);
    if(argc > 2) sscanf(argv[2],"%le",&data2);
    if(argc > 3) sscanf(argv[3],"%le",&data3);
    mgk_write_node_input_port(&data1,T_DOUBLE,n1,0);
    mgk_write_node_input_port(&data2,T_DOUBLE,n1,1);
    mgk_write_node_input_port(&data3,T_DOUBLE,n2,1);
    while(mgk_run(1)) ;
       dp = mgk_peek_node_output_port(n2,0,0,0,&tp);
       if(dp && (tp == T_DOUBLE)) {printf("The result is %f\n",*dp);}
       return(0);
    }
    return(-1);
}
```

References

1. J. Sztipanovits and G. Karsai: Model-Integrated Computing, IEEE Computer, p 110, April, 1997.
2. Karsai,G.,Misra,A.,Sztipanovits,J,Ledeczi,A.,Moore,M.: "Model-Integrated System Development: Models, Architecture, and Process", in Proceedings of the COMPSAC, pp 176-81, 1997.
3. Ledeczi,A.,Maroti,M.,Bakay,A., Karsai,G., Garrett,J., Thomason,C., Nordstrom,G., Sprinkle, J., Volgyesi, P.:"The Generic Modeling Environment". IEEE Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.
4. Nordstrom,G., Sztipanovits,J. Karsai, G: "Meta-level Extension of the Multigraph Architecture", Proc. of the ECBS-98 Conference, pp. 61-68, Jerusalem, Israel, March, 1998.
5. Nordstrom,G.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments, Ph. D. Dissertation, Vanderbilt University, Electrical Engineering and Computer Science, March, 1999.
6. Nordstrom G., Ledeczi A.: Formalizing the Specification of Graphical Modeling Languages, ISIS Technical Report ISIS-00-200, 2000.
7. http://www.isis.vanderbilt.edu/Projects/gme/default.html
8. http://www.isis.vanderbilt.edu/Projects/gme/meta.html
9. Gamma,E., Helm,R.,Johnson,R.,Vlissides,J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
10. Lieberherr,K.: Adaptive Object-Oriented Software:The Demeter Method with Propagation Patterns,PWS Publishing Company, Boston,1996.
11. Karsai, G. "Structured Specification of Model Interpreters", Proc. of the ECBS-99 Conference, pp. 84-91, Nashville, TN, 1999.
12. Biegl, Cs.: Multigraph Kernel Version 6.0 User's Manual.
13. Franke,H.: Evaluation of the Multigraph Kernel on an IBM SP2 Workstation Cluster, Interim Report.
14. Long,E. Misra,A., Sztipanovits,J.: "Increasing Productivity at Saturn", IEEE Computer, pp 35-43, Aug. 1998.
15. Long,E. Misra,A., Sztipanovits,J.: "Saturn Site Production Flow (SSPF): Accomplishments and Challenges", IEEE Conference on Engineering of Computer Based Systems, pp 248-255, Jerusalem, Israel, March, 1998.
16. ITIS Technical Overview, Interim Report, 1999.
17. ITIS Architecture, Technical Report, 1999.
18. Davis J., Scott J., Sztipanovits J., Karsai G., Martinez M.: An Integrated Multi-Domain Analysis Environment for High Consequence Systems, Proceedings of the 1998 ASME Design Engineering Technical Conference / Computers in Engineering, Paper No. 5532, Altanta, GA, September, 1998.
19. Karsai G., Gray J.: Component Generation Technology for Semantic Tool Integration, Proceedings of the IEEE Aerospace 2000, CD-Rom Reference 10.0303, Big Sky, MT , March, 2000.
20. Karsai G.: Design Tool Integration: An Exercise in Semantic Interoperability, Proceedings of the IEEE Engineering of Computer Based Systems, , Edinburgh, UK, March, 2000.
21. Karsai G., Nordstrom G., Ledeczi A., Sztipanovits J.: Towards Two-Level Formal Modeling of Computer-Based Systems, Journal of Universal Computer Science, Vol. 6, No. 11, pp. 1131-1144, November, 2000.

Self-Adaptive Software Project
Executive Summary

The research conducted under the Model-Integrated Architecture for Self-Adaptive Software project is a continuation of our work on Model-Integrated Computing (MIC) and structurally adaptive signal processing and control systems. In MIC, domain specific, multiple-view models represent the computer application, its environment and their relationships. Model interpreters translate the models into the input languages of static and dynamic analysis tools, and application-specific model interpreters synthesize and re-synthesize software applications running in a real-time, dynamic, macro-dataflow execution environment.

The model-integrated approach to self-adaptive software decomposes the problem into two major issues: (1) representation and (2) the reconfiguration mechanism. The representation issue deals with modeling of self-adaptive systems, including models of architectures and adaptation processes. The reconfiguration mechanism focuses on methods for mapping the models into executable systems, and changing the dataflow and control structure of the application in a safe, consistent manner.

Our approach is based on a performance evaluation → architecture modification cycle. In this paradigm, the performance of the application (and/or its components) is continuously monitored, with the results used for calculating modifications to the architecture model (while taking into consideration current resource and consistency constraints). The modification is followed by a partial or complete regeneration of the executable system.

1. System Representation

The need for modeling dynamic architectures is related to the complexity of the system being modeled. In a simplistic approach, one can pre-design all the possible architectures of a system, model all the architectures discovered, assign them to predefined situations, and switch between these architectures as the system evolves. A more sophisticated approach is to structure the architecture representation into a hierarchy with alternatives on each level. In a hierarchically organized architecture description, components contain other components that specify the internal architecture of the parent component in terms of the lower level components and their connectivity. This representation technique scales much better, but still requires that the configuration alternatives be explicitly defined at design time.

A different approach is to represent dynamic architectures in a generative manner. Here, the components of the architecture are prepared, but their number and connectivity patterns are not fully defined at design time. Instead, a generative description is provided which specifies how the architecture could be generated "on-the-fly". A generative architecture specification is similar to the generate statement used in VHDL: it is essentially a program that, when executed, generates an architecture by instantiating components and connecting them together.

The generative description is especially powerful when it is combined with architectural parameters and hierarchical decomposition. In a component one can generatively represent an architecture, and the generation "algorithm" can receive architectural parameters from the current or higher levels of the hierarchy. These parameters influence the architectural choices made (e.g. how many components to use, how they are connected, etc.), but might also be propagated downward in the hierarchy to components at lower levels. There the process is repeated: architectural choices are made, components are instantiated and connected, and possibly newly calculated parameters are passed down further. Thus, with very few generative constructs one can represent a wide variety of architectures that would be very hard, if not impossible, to pre-enumerate.

Naturally, not every architectural alternative is viable in all circumstances. The generative description allows for representing architectural constraints that constrain the selection process, thus limiting the search needed while forcing the process to obey other requirements.

2. Architecture

The basic structure of the proposed self-adaptive embedded model-integrated system is illustrated in the figure below. The Embedded Modeling Infrastructure (EMI) can be best viewed as a high-level layer at the top of the architecture, while a classical embedded system kernel (e.g. a real-time kernel) is located at the bottom. The component that matches these two layers is the translator that we call embedded interpreter. The embedded models provide a simple, uniform, paradigm-independent and extensible API towards this interpreter.

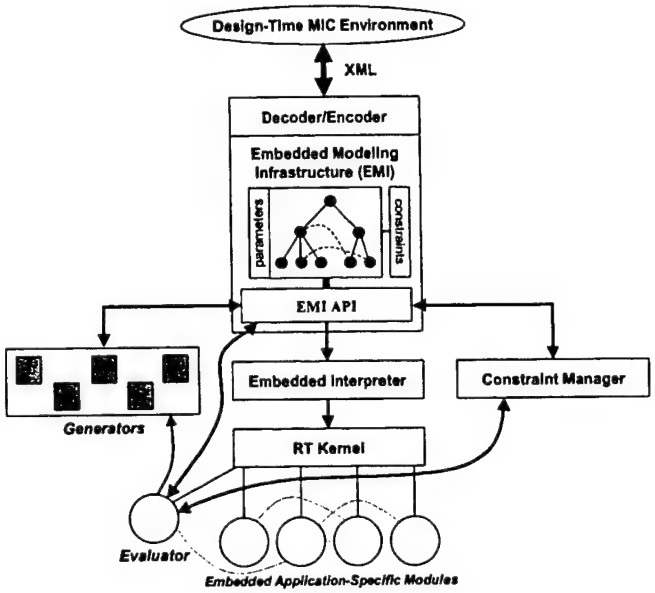

Figure 14: Self-Adaptive MIC Architecture

We define a self-adaptive computing system as an environment that is able to measure and detect changes in its own performance and perform structural changes in its configuration. According to this definition, an adaptive system must contain monitoring, decision making, and configuration modification functionality. The EMI system provides capabilities for each of these three tasks.

Monitoring requires the presentation of operational parameters in a convenient fashion. Operational parameters in embedded systems can be divided into two distinct categories:

- The momentary status of the embedded operating system itself: resource utilization, timing relations, availability of peripherals and remote nodes in a distributed system, etc.
- The parameters representing the performance of the application algorithms, such as the control error rates of a control system, the amount of missed/discarded packets of a communication system, cost functions, etc. A common feature of these parameters is that they are application-specific, and their values are usually highly dependent on the environment of the embedded system.

Our decision was to use the embedded models as a place for uniform representation of these parameters. Objects in the model may have designated attributes (monitor attributes) set by the underlying modules: either by the embedded kernel (in case of most operation system

parameters), or by any of the application specific task modules that have information available on the operation of the algorithm.

The task listed third, configuration modification, is supplied by generative models and generators. This way, the model designer can efficiently control the degrees of freedom in the model by providing generators where adaptive modifications in the model are foreseen. This decision also reflects the fact that self-adaptive modifications are usually similar or identical to alterations executed by an external management system (or a human operator) on a non-self-adaptive system. Model changes made by the generators are translated towards the kernel by the embedded interpreter.

The most critical part of self-adaptive applications is the one making reconfiguration decisions. In our embedded infrastructure, this is the evaluator, a kernel process itself, which is responsible for interpreting the monitor parameters and setting architectural parameters for the generators. It is obvious that depending on the complexity of the application domain and level of adaptivity implemented, the knowledge of such an evaluator may range from some simple mapping operations to real intelligence comparable to that of a human expert. This also means that the programming model may vary in a wide range: data tables, procedural code, data-flow network, or some more esoteric techniques, such as genetic algorithms or neural nets.

These considerations led us to a decision to leave the selection of the evaluator to the application designer: from the EMI system's point of view, the evaluator is a native module utilizing to the standard EMI API, and expected to take input mostly from the monitor parameters in the model, and produce its outputs by setting the values of the generator parameters.

In this architecture, the EMI has the following functions:

- Loading the initial version of the model from an external source (typically a modeling/management computer) or from some internal storage facility,
- Booting the embedded system based on the models loaded. This includes executing the built-in generators, which, in-turn, create the dynamic parts of the models through the embedded interpreter,
- Checking model constraints and implementing emergency measures in case of failures,
- Evaluating the operation of the embedded programming modules and setting generative parameters accordingly. Again, the model generators have the task to implement these changes on the models themselves,
- Receiving and executing model updates from external sources, and
- Communicating status information to external management agents.

During the load/boot phase the EMI builds the data structures of the initial model and, based on this information, instructs the kernel to instantiate the computing objects accordingly. Generators execute and generate the structure that corresponds to the initial status of the generative parameter objects. Generators are hybrid objects. While they are part of the embedded models, the compiled object code of the generator scripts cannot be represented as paradigm independent models in the EMI. These scripts, therefore, are located outside of the EMI. Note that they use the same EMI API as, for example, the embedded interpreter. From a programmers point of view, generative modeling and interpreter implementation are similar activities.

The Constraint Manager is a module responsible for ensuring that modeling constraints are met. There are two principal questions about constraints: when to check them, and what to do when (resulting from modifications by the self-adaptive control loop or external modification commands) they are violated. The answer to these questions is highly application dependent, therefore, the Constraint Manager is a designed to be a flexible component with its own API. The evaluator, for example, can request constraint checking for any subset of constraints and/or models at any time.

3. Constraint-Guided Self-Adaptation

The above technique is a practical approach to creating self-adaptive embedded systems. It's main limitation is that it does not address directly one of the most challenging aspects of self-adaptivity: the *evaluator*, the component who decides when and how to reconfigure the system. It does provide an infrastructure for the user to insert the evaluator component into the overall system, but the its functionality needs to be implemented by the user by hand. Recently, we have explored a systematic approach to address this problem.

The approach builds on the Embedded Modeling Infrastucture (EMI) as well, but instead of using generative models, the embedded models of the system capture multiple potential system configurations as alternatives in the operation space. Parameterized constraints provide a way of capturing changing operational requirements. An embedded operation space exploration tool, triggered by changes in operational parameters, rapidly finds the next system configuration that satisfies the current operational constraints, which is then instantiated and deployed through the EMI.

The main contribution of the work is the systematic approach to the evaluator, the key component of any self-adaptive system. The OBDD-based symbolic satisfaction of constraints parameterized by monitored operational variables makes the controller reusable across applications and even application domains. It can potentially replace the ad-hoc, highly application-specific, hand-crafted reconfiguration controllers of the past. Another principal benefit of this approach is that the every system configuration that is deployed is correct by design, i.e. it already satisfies all the correctness criteria specified by the constraints.

An ideal application area for our approach is the domain of robust, fault-tolerant embedded systems. The operation space can contain various system configurations that are tailored for different system failure modes. Constraints capture the complex relationships between different failure modes and system configurations. The system can quickly find a new configuration and adapt after a component failure.

There are numerous open research issues associated with the approach and a lot of work remains to be done. The most difficult problem is the operation-space exploration within strict time bounds and utilizing possibly limited computational resources. A systematic approach to over- or under-constrained systems needs to be developed. Finally, the technique needs to be implemented and tested in real-world scenarios.

4. Publications

The following publications document our research results in much more detail. The papers marked by the → symbol are attached to this report.

- → Ledeczi A., Bakay A., Maroti M.: "Model-Integrated Embedded Systems," in Robertson, Shrobe, Laddaga (eds) *Self Adaptive Software*, Springer-Verlag Lecture Notes in CS, #1936, 2001
- → Neema, S., Ledeczi, A.: "Constraint Guided Self-Adaptation," in the Proceedings of the International Workshop on Self-Adaptive Software, Balatonfured, Hungary, May 2001
- → Karsai G., Ledeczi A., Sztipanovits J., Peceli G., Simon G., Kovacshazy T.: "An Approach to Self-Adaptive Software based on Supervisory Control," in the Proceedings of the International Workshop on Self-Adaptive Software, Balatonfured, Hungary, May 2001
- → Bakay, A.: "Model-Based Adaptivity in Real-Time Scheduling," in the Proceedings of the International Workshop on Self-Adaptive Software, Balatonfured, Hungary, May 2001
- → Ledeczi, A. et al.: "Synthesis of Self-Adaptive Software," in the Proceedings of the IEEE Aerospace Conference, March 2000
- Karsai, G., Sztipanovits, J.: "A Model-Based Approach to Self-Adaptive Software," in *IEEE Intelligent Systems*, May/June 1999

- Sztipanovits, J., Karsai, G., Bapty, T.: "Self-Adaptive Software for Signal Processing," in *Communications of the ACM*, May 1998

Professional Personnel Associated with the Project

Dr. Janos Sztipanovits, PI

Dr. Gabor Karsai, Co-PI then PI

Dr. Akos Ledeczi, Co-PI then PI

Dr. Ted Bapty, Co-PI

Dr. Csaba Biegl, Research Scientist

Dr. Arpad Bakay, Research Scientist

Mr. Joe Christopher, Research Programmer

Mr. Larry Howard, Research Programmer

Mr. Dinesh Deva, Research Assistant, MSc degree completed 2001

Mr. Miklos Maroti, Research Assistant

Mr. Greg Nordstrom, Research Assistant, PhD degree completed 1999.

Mr. Surya Pathak, Research Assistant. MSc degree completed 2001

Mr. Stephan Rosner, Research Assistant. MSc degree completed 1998

Mr. Wesley Williams, Research Assistant

Cumulative List of Publications related to the Project

These publications are available on the ISIS website: www.isis.vanderbilt.edu

Technical Reports:

- Ledeczi A., Maroti M., Bakay A., Nordstrom G., Garrett J., Thomason IV C., Sprinkle J., Volgyesi P.: GME 2000 Users Manual (v1.1), document, April 12, 2001.

- Nordstrom G., Ledeczi A.: Formalizing the Specification of Graphical Modeling Languages, ISIS Technical Report ISIS-00-200, 2000.

Journal Articles:

- Ledeczi A., Bakay A., Maroti M.: Model-Integrated Embedded Systems, in Robertson, Shrobe, Laddaga (eds) Self Adaptive Software, Springer-Verlag Lecture Notes in CS, #1936, February, 2001.

- Karsai G., Nordstrom G., Ledeczi A., Sztipanovits J.: Towards Two-Level Formal Modeling of Computer-Based Systems, Journal of Universal Computer Science, Vol. 6, No. 11, pp. 1131-1144, November, 2000.

- Sztipanovits J., Karsai G.: A Model-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems, 14, 3, pp. 46-53, 1999.

- Long E., Misra A., Sztipanovits J.: Increasing Productivity at Saturn, IEEE Computer Magazine, August, 1998.

- Sztipanovits J., Karsai G.: Self-Adaptive Software for Signal Processing, CACM, 41, 5, pp. 55-65, 1998.

- Sztipanovits J., Karsai G.: Model-Integrated Computing, IEEE Computer, pp. 110-112, April, 1997.

Conference Proceedings:

- Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom, G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, accepted, Budapest, Hungary, May 17, 2001.

- Karsai G., Ledeczi A., Sztipanovits J., Peceli G., Simon G., Kovacshazy T.: An Approach to Self-Adaptive Software based on Supervisory Control, IWSAS-2001, (submitted), Balatonfured, Hungary, May, 2001.

- Sprinkle J., Karsai G., Ledeczi A., Nordstrom G.: The New Metamodeling Generation, IEEE Engineering of Computer Based Systems, Proceedings p.275, Washington, D.C., USA, April, 2001.

- Karsai G., Nordstrom G., Ledeczi A., Sztipanovits J.: Specifying Graphical Modeling Systems Using Constraint-based Metamodels, IEEE Symposium on Computer Aided Control System Design, Conference CD-Rom, Anchorage, Alaska, September 25, 2000.

- Nordstrom G., Karsai G., Moore M., Bapty T., Sztipanovits J.: Model Integrated Computing-Based Software Design and Evolution, Conference on Life Cycle Software

Engineering Technology for Modern Avionics, Missiles, and Smart Weapon Systems, , Huntsville, Alabama, August 16, 2000.

- Sztipanovits J., Karsai G., Franke H.: Model-Integrated Program Synthesis Environment, Proceedings of the IEEE Symposium on Engineering of Computer Based Systems, pp. 348-355, Friedrichshafen, Germany, March 11, 1998.

- Karsai G., Gray J.: Component Generation Technology for Semantic Tool Integration, Proceedings of the IEEE Aerospace 2000, CD-Rom Reference 10.0303, Big Sky, MT , March, 2000.

- Nordstrom G.: Formalizing the Specification of Graphical Modeling Languages, Proceedings of the IEEE Aerospace 2000 Conference, CD-ROM Reference 10.0402, Big Sky, MT, March, 2000.

- Karsai G.: Design Tool Integration: An Exercise in Semantic Interoperability, Proceedings of the IEEE Engineering of Computer Based Systems, , Edinburg, UK, March, 2000.

- Ledeczi A., Bapty T., Karsai G.: Synthesis of Self-Adaptive Software, IEEE Aerospace 2000 , CD-ROM Reference 10.0304, Big Sky, MT, March, 2000.

- Ledeczi A.: Model Construction for Model-Integrated Computing, 13th International Conference on Systems Engineering, pp. CS103-108, Las Vegas, NV, August, 1999.

- Nordstrom G., Sztipanovits J., Karsai G., Ledeczi A.: Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments, Proceedings of the IEEE ECBS'99 Conference, pp. 68-74, Nashville, Tennessee, April, 1999.

- Ledeczi A., Maroti M., Karsai G., Nordstrom G.: Metaprogrammable Toolkit for Model-Integrated Computing, Engineering of Computer Based Systems (ECBS) , pp. 311-317, Nashville, TN, March, 1999.

- Karsai G.: Structured Specification of Model Interpreters, ECBS, pp 84-91, Nashville, TN, March, 1999.

- Davis J., Scott J., Sztipanovits J., Martinez M.: Multi-Domain Surety Modeling and Analysis for High Assurance Systems, Proceedings of the Engineering of Computer Based Systems, pp. 254-260, Nashville, TN , March, 1999.

- Davis J., Scott J., Sztipanovits J., Karsai G., Martinez M.: An Integrated Multi-Domain Analysis Environment for High Consequence Systems, Proceedings of the 1998 ASME Design Engineering Technical Conference / Computers in Engineering, Paper No. 5532, Altanta, GA, September, 1998.

- Nordstrom G., Sztipanovits J., Karsai G.: Metalevel Extension of the MultiGraph Architecture, Proceedings of the IEEE ECBS'98 Conference, pp. 61-68, Jerusalem, Israel, April, 1998.

- Davis J., Scott J., Sztipanovits J., Karsai G., Martinez M.: Integrated Analysis Environment for High Impact Systems, Proceedings of the Engineering of Computer Based Systems, pp. 218-225, Jerusalem, Israel, April, 1998.

- Long E., Misra A., Sztipanovits J.: Saturn Site Production Flow (SSPF): Accomplishments and Challenges, Engineering of Computer Based Systems, pp 248-255, Jerusalem, Israel, March, 1998.

- Franke H., Biegl C.: Evaluation of a Graph Computational Model on an IBM SP2 Scalable Workstation Cluster, Proceedings of the 1998 IEEE Conference on Engineering of Computer-Based Systems, pp 166-173, Jerusalem, FC, March, 1998.

- Karsai G., Sztipanovits J., Franke H.: Towards Specification of Program Synthesis in Model-Integrated Computing, ECBS-98, pp 226-233, Jerusalem, Israel, 1998.

- Franke H., Sztipanovits J., Karsai G.: Model-Integrated Computing, Hawaii Systems of the World Manufacturing Congress, CD-ROM publication, Auckland, New Zealand, November, 1997.

- Misra A., Long E., Sztipanovits J.: Evolutionary Design for Manufacturing Execution Systems, World Manufacturing Congress, , Auckland, New Zealand, November, 1997.

- Karsai G., Sztipanovits J., Ledeczi A., Moore M.: Model-Integrated System Development: Models, Architecture and Process, 21st Annual International Computer Software and Application Conference (COMPSAC), pp. 176-181, Bethesda, MD, August, 1997.

- Misra A., Karsai G., Sztipanovits J.: Model-Integrated Development of Complex Applications, Fifth International Symposium on Assessment of Software Tools, pp 14-23, Pittsburgh, PA, June, 1997.
- Misra A., Karsai G., Sztipanovits J., Ledeczi A., Moore M.: A Model-Integrated Information System for Increasing Throughput in Discrete Manufacturing, International Conference and Workshop on Engineering of Computer Based Systems, pp 203-210, Monterey, CA, March 24, 1997.
- Ledeczi A.: Model-Integrated Parallel Application Synthesis, Engineering of Computer Based Systems (ECBS) , pp. 38-45, Monterey, CA, March, 1997.

- Bapty T., Sztipanovits J.: Model-Based Engineering of Large-Scale Real-Time Systems, Proceedings of the Engineering of Computer Based Systems (ECBS) Conference, pp. 467-474, Monterey, CA, March, 1997.

- Long E., Misra A.: A Model-Based Engineering Process for Increasing Productivity in Discrete Manufacturing, International Conference and Workshop on Engineering of Computer Based Systems, pp 197-202, Monterey, CA, March, 1997.

Academic Papers:

- Davis J.: Integrated Safety, Reliability, and Diagnostics of High Assurance, High Consequence Systems, Ph.D. Dissertation, Vanderbilt University, Electrical Engineering, 2000.

- Wang J.: Visual Specification of Model Interpreters, Master's Thesis, Vanderbilt University, 2000.

- Nordstrom G.: Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments, Ph. D. Dissertation, Vanderbilt University, Electrical Engineering and Computer Science, March, 1999.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.